Santos *et al. Journal of Software Engineering Research and Development* 2014, **2**:11
www.jserd.com/content/2/1/11

Journal of Software Engineering Research and Development
a SpringerOpen Journal

RESEARCH Open Access

The problem of conceptualization in god class detection: agreement, strategies and decision drivers

José Amancio M Santos[1*†], Manoel Gomes de Mendonça[2,3†], Cleber Pereira dos Santos[4†] and Renato Lima Novais[4†]

*Correspondence:
zeamancio@ecomp.uefs.br
†Equal contributors
[1]Department of Technology, State University of Feira de Santana, Transnordestina avenue S/N - Feira de Santana - Bahia, Feira de Santana, Brazil
Full list of author information is available at the end of the article

Abstract

Background: The concept of code smells is widespread in Software Engineering. Despite the empirical studies addressing the topic, the set of context-dependent issues that impacts the human perception of what is a code smell has not been studied in depth. We call this the code smell *conceptualization problem*. To discuss the problem, empirical studies are necessary. In this work, we focused on conceptualization of god class. God class is a code smell characterized by classes that tend to centralize the intelligence of the system. It is one of the most studied smells in software engineering literature.

Method: A controlled experiment that extends and builds upon a previous empirical study about how humans detect god classes, their decision drivers, and agreement rate. Our study delves into research questions of the previous study, adding visualization to the smell detection process, and analyzing strategies of detection.

Result: Our findings show that agreement among participants is low, which corroborates previous studies. We show that this is mainly related to agreeing on what a god class is and which thresholds should be adopted, and not related to comprehension of the programs. The use of visualization did not improve the agreement among the participants. However, it did affect the choice of detection drivers.

Conclusion: This study contributes to expand empirical evidences on the impact of human perception on detecting code smells. It shows that studies about the human role in smell detection are relevant and they should consider the conceptualization problem of code smells.

Keywords: Code smell; God class; Controlled experiment; Code visualization

Background

Challenges in object-oriented (OO) software design have been historically addressed from different perspectives. Riel (1996) wrote one of the first books on the subject in 1996. This book presents insights into OO design improvements and introduces the now well-known term "design flaws". In 1999, Fowler (1999) came up with the concept of refactoring and coined the term "smell" to represent bad characteristics observable in the code. In 2005, Lanza and Marinescu (2005) focused on OO metrics to characterize what they

called "disharmonies". All these terms are used to define potential design problems. In this paper, we adopt the term *code smell*, or simply *smell*, to refer to such design problems.

The works of (Riel 1996; Fowler 1999 and Lanza and Marinescu 2005) discuss code smells from the principles of the OO paradigm, such as information hiding or polymorphism (Meyer 1988). However, there is a set of context-dependent issues that impacts how one considers the concept of smell. These include: developers' experience, the software process, software domain, and others. The extensive number of context-dependent issues make it difficult to express even simple tasks rigorously, such as smell detection. Fontana et al. (2011) claim that smell detection "can provide uncertain and unsafe results". This is because most smells are subjectively defined and their identification is human-dependent.

Fowler (1999) does not define smell formally. He says that one needs to develop one's own sense of observation of attributes that could characterize pieces of code as a smell. For example, one has to develop one's own sense of how many lines of code define a long method. This is because smell detection is a subjective task by nature. In contrast, Lanza and Marinescu (2005) use a formal definition for smells based on metrics and thresholds. However, Rapu et al. (2004) state that the thresholds are mainly chosen based on the experience of the analysts. These indicate that smell detection remains an ill-defined task. In addition, as cited by Parnin et al. (2008), "metrics produce voluminous and imprecise results". Given these pitfalls, alternatives have emerged to address smell detection. One of them is the use of software visualization (Murphy-Hill and Black 2010; Parnin et al. 2008; Simon et al. 2001; Van Emden and Moonen 2002). Software visualization tools combined with metrics may help humans to identify design problems.

Understanding which, and how, subjective aspects affect smell detection demands empirical evaluation. According to Mäntylä, "we need more empirical research aiming at critically evaluating, validating and improving our understanding of subjective indicators of design quality" (Mäntylä et al. 2004). Recent empirical studies carried out to better understand this scenario, can be classified into three categories. The first type is correlation studies. They evaluate the impact of smells based on data extracted from software repositories (Li and Shatnawi 2007; Olbrich et al. 2009, 2010), establishing a correlation between a smell and some attribute of the software, such as bugs or the number of modifications on classes. The second type is related to tool assessment, such as automatic detection (Moha et al. 2010; Mäntylä and Lassenius 2006a; Schumacher et al. 2010) or software visualization (Carneiro et al. 2010; Murphy-Hill and Black 2010; Parnin et al. 2008; Simon et al. 2001). Finally, the third type investigates the role of humans in smell detection (Mäntylä 2005; Mäntylä and Lassenius 2006b; Santos et al. 2013; Schumacher et al. 2010). Although the number of studies on this topic is increasing, they are considered insufficient (Schumacher et al. 2010; Sjøberg et al. 2013; Zhang et al. 2011). In particular, the role of humans has not been studied in depth (Mäntylä and Lassenius 2006a).

The role of humans is one of the most important and one of the most open and broad topics in this area. Several uncontrollable variables affect the smell conceptualization. Examples include experience, personal differences in cognition, level of knowledge on the subject, and the environment. In this context, this work aims to understand how some aspects of the human conceptualization impact smells detection. In particular, this exploratory study investigates how personal comprehension of the smell concept affects the detection of god classes. God class is a term proposed by Riel (1996) to refer to classes

that tend to centralize the intelligence of the system. Fowler (1999) defined it as a class that tries to do too much, and adopted the term large class. Lanza and Marinescu (2005) proposed a heuristic based on metrics to identify god class. The definition of god class has been addressed in several empirical studies (Abbes et al. 2011; Li and Shatnawi 2007; Olbrich et al. 2010; Padilha et al. 2013).

Our exploratory study was based on a controlled experiment carried out in an in-vitro setting. The experiment extended an empirical study presented by Schumacher et al. in (Schumacher et al. 2010). While Schumacher et al. presented a wide discussion about both human and automatic detection of god classes, our work focused on questions related to the human perception. We considered two factors, the use or non use of software visualization to achieve the study goal. The use of visualization made it possible to evaluate the impact of the overall comprehension of the design on human aspects, increasing the evidences of the relevance of conceptualization in the detection of god class. It is important to note that we did not adopt a visualization tool to support smell detection, instead, we adopted a tool focused on enhancing the comprehension of the code design. The tool helps developers to perceive of coupling, size, complexity, and hierarchical relations among classes from the use of visual resources. We evaluated the effect of these facilities on human aspects, such as decision drivers, agreement and strategies adopted by the participants detecting god classes. To the best of our knowledge, this is the first study that analyses smell conceptualization using all of these variables. We have already featured this experiment partially (Santos et al. 2013), addressing effort, decision drivers, and agreement on smell detection, but disregarding the use of visualization.

The structure of this paper is as follows. Section 'Method' presents the planning and execution of the experiment. Section 'Results' and 'Discussion' present the results and a discussion about them. Section 'Threats to validity' discusses the threats to the validity of the study. Section 'Context and related works' summarizes prior empirical studies that address aspects which are context-related to smells. Lastly, Section 'Conclusions' presents our conclusions and proposes future works.

Method

In this section, we present the experimental planning and execution of the experiment.

In order to attend ethical issues on Empirical Software Engineering, we followed principles proposed by (Vinson 2008).

Research question

Our work aims to investigate the impact of conceptualization on god class detection. The research questions (RQ) are:

1. *How well do humans agree on identifying god classes?*
2. *How well do humans and an oracle agree on identifying god classes?*
3. *Which strategies are used to identify god classes?*
4. *What issues in code lead humans to identify a class as a god class?*

All questions help us to observe the differences in perception among the participants during the detection of god classes. They were used to observe how conceptualization affects the identification of god classes in our experiment. Research questions one and four were first proposed by Schumacher et al. (2010). In this paper, we analyzed them

using a different approach: by the use of both code review and visualization. We have introduced question two and three. Question two addresses agreement between participants and an oracle, defined in a controlled process. Question three addresses the strategies adopted by participants when detecting god classes.

Experimental units

The experiment involved 11 undergraduate Computer Science students from the Federal University of Bahia (UFBA), in Brazil. All students were enrolled in the Software Quality course offered in the first semester of 2012. This is an optional subject of the Computer Science program, in which design quality and smells are addressed. The course was considered appropriate for the experiment, both because it was focused and was not mandatory, which means that most students enrolled on it were interested in the subject. Furthermore, participation in the experiment was a voluntary activity.

Experimental material

Tools

We adopted four software tools in the experiment[a]: (i) The Eclipse Indigo IDE; (ii) Usage Data Collector (UDC), an Eclipse plug-in for collecting IDE usage data information (interactions between participants and Eclipse can be accessed by the log of UDC). This tool is embedded in the Eclipse Indigo IDE; (iii) Task Register plug-in, a tool we developed to enable participants to indicate what task was being done at any given moment. This information was also registered in the UDC log. All the participants had to do was to click on a "Task Register" view on Eclipse (Figure 1-F) to indicate when they were starting or finishing a task; and (iv) SourceMiner, an Eclipse plug-in that provides visual resources to enhance software comprehension activities (Carneiro and Mendonça 2013; Carneiro et al. 2010).

SourceMiner has five views, divided into two groups. The first group is made up of three coupling views. These views show different types of dependencies among entities, like direct access to attributes or method calling, for instance. Moreover, they show the

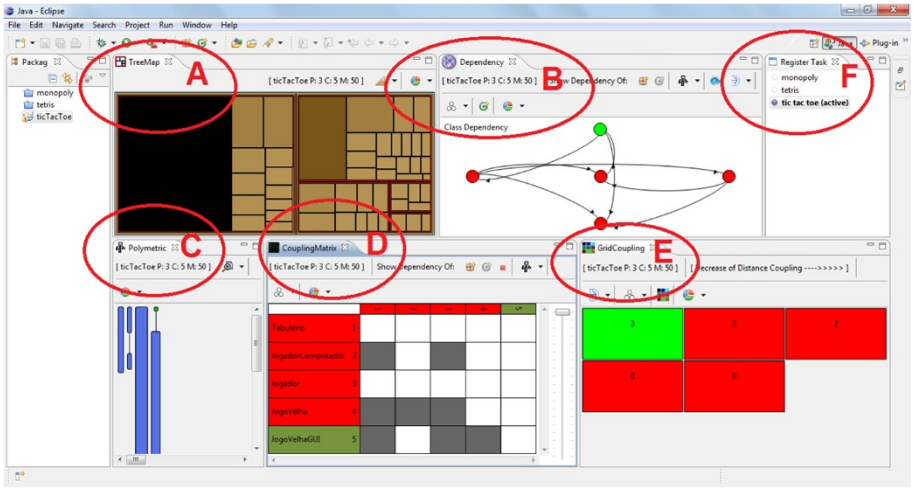

Figure 1 Eclipse IDE showing the Task Register (F) plug-in and SourceMiner views: hierarchical views (A and C) and coupling views (B, D and E).

direction of the coupling. The coupling views are based on radial graphs (Figure 1-B), matrix of relationships (Figure 1-D), and tabular view (Figure 1-E). The second group is made up of two hierarchical views. These views associate the number of lines of code, complexity and number of methods of classes to the area and colors of the rectangles. The Treemap view (Figure 1-A) shows the hierarchy of package-class-method of the software. A Treemap is a hierarchical 2D visualization that maps a tree structure into a set of nested rectangles (Johnson and Shneiderman 1991). In SourceMiner, the nested rectangles represent software entities, like packages, classes and methods. Rectangles representing methods of the same class are drawn together inside the rectangle of the class. Likewise, the rectangles of the classes that belong to the same package are drawn together inside the rectangle of the package. The Polimetric view (Figure 1-C) shows the hierarchy between classes and interfaces. A polymetric view uses a forest of rectangles to represent the inheritance trees formed by classes and interfaces in a software system (Lanza and Ducasse 2003). Rectangles are linked by edges representing the inheritance relationship between them. The length and width of the rectangles can be used to represent software attributes such as the size and number of methods of a class.

Forms

We used five forms and two guides during the experiment. During the training, which we will present thereafter, the participants filled in a Consent and a Participant Characterization form, and received a SourceMiner exercise guide. During the experiment itself, participants received a Support Question guide to steer them in search for god classes. The questions are the same ones used by Schumacher et al. (2010):

- Does the class have more than one responsibility?
- Does the class have functionality that would fit better into other classes?
  - By looking at the methods, could one ask: "Is this the class' job?"
- Do you have problems summarizing the class' responsibility in one sentence?
- Would splitting up the class improve the overall design?

Another document used during the experiment was the Step-by-Step guide used to assure consistency during the data collection process. This document prompts the participants to open and close the projects on Eclipse and to select the task under execution in the Task Register view. It is important to note that this guide did not define how participants should do the task of identifying god classes. It defined some activities that participants had to do before and after the identification of god classes. Each participant defined their own strategy to identify god classes.

The Answer form was one of the most important forms used during the experiment. On this form, participants had to fill in: i) one or more candidate(s) god class(es), ii) their level of certainty, i.e., fill in the "yes" or "maybe" option for each candidate class, iii) the decision driver(s) which helped her/him to select the class, and iv) the start and end time of the task, which we used to evaluate effort in (Santos et al. 2013) (this topic is out of the scope of this paper).

We considered the items i) the candidate god classes and ii) level of certainty, which is self-explanatory. Here, we will explain item iii) decision drivers: the form listed nine drivers as predefined options for the participants, but it was also possible to write down a

new one. The drivers listed in the form are the ones identified by Schumacher et al. (2010) during their think-aloud data collection. Some examples are "method is misplaced" and "class is highly complex".

The last form of FinG was the Feedback form. At the end of the experiment the participants filled in a Feedback form. On it, we asked the participants to classify the training and the level of difficulty performing the task. It was also possible to write down suggestions and observations about the experiment.

Software artifacts

Six programs were used in the experiment. All of them implement familiar applications or games in Java e.g. Chess, Tic Tac Toe, Monopoly and Tetris. Solitaire-Freecell (Solitaire) is a framework for card games with Solitaire and Freecell. Jackut implements a simple social network application, such as Facebook and Orkut.

During our selection, we looked for familiar applications to minimize the effort of participants during the task of identifying god classes. However, we also looked for programs with different characteristics: some without god classes, others which perhaps had god classes and others with, at least, one god class. An oracle was used to identify god classes in each of the selected programs. This oracle is presented later. Table 1 characterizes the programs we used in terms of the number of packages, number of classes and number of lines of code (LOC).

Task

While Schumacher et al. (2010) designed a mini-process for participants to detect god classes, we gave only support questions as a guide. Each participant was free to use her/his own strategy to do the task. Furthermore, they could choose the order of the tasks. The Step-by-Step guide only indicated the activities to be done before and after the god class detection for each program, such as filling in the start and finish time on the answer form.

Design

The experiment was carried out in a laboratory at UFBA. Participants had 2 hours to carry out the task. Each participant worked at a separate workstation. At each workstation, we set up two Eclipses IDEs. One contained the SourceMiner plug-in, and the other did not. Both Eclipses were fitted with a Task Register and UDC plug-ins. Each Eclipse had three of the six programs in their workspace. The workstations were divided into two groups. After analysis of the characterization form, participants were randomly allocated to groups, because they had similar profiles: years of programming and knowledge on the topic were used. There were six participants in group 1 and five participants in group 2. We present the distribution of participants by group in Table 2. The ID of the participants was formed by the position of the workstation in the lab where the task was performed.

Table 1 Software objects

Software	Chess	Jackut	Tic Tac Toe	Monopoly	Solitaire	Tetris
Packages	5	8	2	3	6	4
Classes	15	19	5	10	23	16
LOC	1426	978	616	2682	1758	993

Table 2 Set of Eclipses installations at the workstations and allocation of participants

Group	With SourceMiner	Without SourceMiner	Participants' ID
1	Chess, Jackut and Solitaire	Monopoly, Tetris and Tic Tac Toe	F14, F21, F32, F35, F42 and F44
2	Monopoly, Tetris and Tic Tac Toe	Chess, Jackut and Solitaire	F13, F15, F25, F31 and F41

Execution

The experiment took four days. Two days were allocated to training, one day for a pilot and one day to perform the experiment. There were three small presentations on the first day of training. As the experiment was a voluntary activity and students had little experience with experimental software engineering, we decided to do a motivational presentation. In this presentation, we discussed the experimental software engineering scene, tying it with discussions about smell effects. The second presentation focused on smells and god class concepts. The third presentation showed the design of the experiment: we just talked about the lab, the individual use of a workstation with two different set ups of the IDE, and the time.

On the second day of training, we did an activity in the lab focusing on the SourceMiner tool (explanation and exercise).

On the third day, we ran a pilot experiment with two students who were also enrolled in the same course of the participants. These two students were out of the 11 we presented in the Table 2. The pilot helped us to evaluate the use of the answer form in paper or electronic format. In the pilot, we presented the Step-by-Step guide in paper format. We did this because we thought that it would not be useful to use electronic format forms because the experiment ran with two opened Eclipses installations. However, after the pilot, we noted that paper forms were less convenient. Therefore, in the final experiment, we kept only the answer sheet form and Support Questions in paper format. The pilot also helped us to validate the inspection time. We confirmed that 1.5 - 2.0 hours was enough time to analyze the six programs.

On the final day we ran the experiment. Table 3 shows the complete schedule. The column Day gives an idea of the time between the activities. For example, the second training (Day 8) was seven days after the first training (Day 1).

Table 3 Experiment schedule

Day	Activity	Presentation	Local	Time (Hour)
1	Training	Motivational + Concepts + Experiment design	Classroom	2,0
8	Training	SourceMiner + Exercise	Lab Lab	2,0
18	Pilot	-	Lab	1,5
20	Experiment	-	Lab	2,0

Deviations

We ran the experiment with 17 students, but only 11 completed the experiment. Four students missed at least one presentation and were excluded. Two students participated in the pilot experiment. We also had an unexpected problem with the schedule. The original schedule was changed and there was a holiday between the SourceMiner presentation and experiment. Due to this, there were only 2 days between the pilot and the experiment. Despite this, the pilot still helped us, as previously discussed.

Data

We collected and analyzed two types of data. The first was the answers on the answer form, as explained in the Section 'Forms': i) the selected god class candidates; ii) the level of certainty, i.e. "yes" or "maybe" option for each candidate class; and iii) decision drivers that helped participants to indicate the candidate god classes.

The other type of data was the UDC Log. We used the UDC plug-in to log participants' actions while the experiment was running. UDC is a framework for collecting usage data on various aspects of the Eclipse workbench. It gathers information about the kinds of activities that the user does in the IDE (i.e. activating views, editors, etc.). The Task Register (Figure 1-F) was used to enrich the UDC log with the name of the program on which the participant was performing the task. Figure 2 shows a clipping of the UDC log annotated by the Task Register plug-in. The first column ("task") does not exist in the original UDC log. It was added by the Task Register. We highlighted columns that we were interested in. The first column ("task") indicates the program for which the participant was doing the god class detection task. Columns "what", "kind" and "description" describe the actions. For example, the first line represents: user activated the Package Explorer view.

Each line represents one user action. As a result, we have sequences of actions for each participant and for each program.

Results

This section presents the results of the experiment. We created one subsection to present the results for each research question.

RQ1: How well do the participants agree on identifying god classes?

To evaluate this research question we considered the agreement on the candidate god classes for both cases, with and without visualization. For the case without visualization, there were six participants in group 1 (Monopoly, Tetris and Tic Tac Toe) and five in group 2 (Chess, Jackut and Solitaire). In the case of with visualization, there were six participants in group 1 (Chess, Jackut and Solitaire) and five participants in group 2 (Monopoly, Tetris and Tic Tac Toe). We tabulated god class candidates into the "yes" and "maybe" category

task	what	kind	bundleId	bundleVersion	description	time
monopoly	activated	view	org.eclipse.jdt.ui	3.7.2.v20120109-1427	org.eclipse.jdt.ui.PackageExplorer	1339695044177
monopoly	opened	editor	org.eclipse.jdt.ui	3.7.2.v20120109-1427	org.eclipse.jdt.ui.CompilationUnitEditor	1339695082575
monopoly	opened	view	org.gesa.sourceminer	2.17.0	org.gesa.sourceminer.view.paradigms.DependencyView	1339695178368
monopoly	activated	view	org.gesa.sourceminer	2.17.0	org.gesa.sourceminer.view.paradigms.CouplingMatrixView	1339695425068
monopoly	executed	command	org.eclipse.ui	3.7.0.v20110928-1505	org.eclipse.ui.views.showView	1339695425069
monopoly	activated	editor	org.eclipse.jdt.ui	3.7.2.v20120109-1427	org.eclipse.jdt.ui.CompilationUnitEditor	1339695457162
monopoly	activated	view	org.eclipse.jdt.ui	3.7.2.v20120109-1427	org.eclipse.jdt.ui.PackageExplorer	1339695457205

Figure 2 Clipping of the user UDC log.

for each participant. Table 4 summarizes the results for the Solitaire program. The F13 participant, for example, marked one class as a "yes" and one class as a "maybe" god class. To analyze the results, we consider both to be god class candidates.

Out of all of the data sets, there were ten cases in which participants filled in the class name and the drivers, but did not mark the option "yes" or "maybe". In these cases, we considered the weakest option, i.e., "maybe". There were also two cases in which the participants did not fill in the name of the class. In this case, we excluded the data entry from the analysis.

We used two approaches to address the research question. The first considered the percentage of candidate classes. The second was an agreement test.

The percentage of god class candidates

To analyze and compare cases with and without visualization, we generated bar chart diagrams (Figure 3). The diagrams show the percentage of candidate god classes with respect to the number of classes for each program. We show the number of classes for each program under the name of the programs. Considering "yes" or "maybe" options, the percentage of god class candidates tends to be higher for programs with a fewer number of classes. In the case of without visualization, the blue bars, there is a small difference for the Tetris program. However the tendency is the same: the percentage of god class candidates tends to be higher for programs with a fewer number of classes. In the case of with visualization, the gray bars, the difference is for the Solitaire program.

Considering only the "yes" option, the difference in the percentage of god class candidates is very small among the programs. Despite this, in the case without visualization, the percentage of god class candidates tends to be higher for programs with a fewer number of classes, once again. For the case with visualization, we noted that the values are very similar, excepted for the Monopoly program, which had the lowest percentage of god class candidates.

Finn agreement test

To evaluate the level of agreement among participants, we adopted the Finn coefficient (Finn 1970) as opposed to the Kappa coefficient (Fleiss 1971), adopted by Schumacher et al. (2010). We did this because of the problems identified in the Kappa coefficient by other authors (Feinstein and Cicchetti 1990; Gwet 2002; Powers 2012; Whitehurst 1984). The Kappa test is done in two phases. First an agreement rate is calculated, and, then this value is used to calculate the coefficient. Feinstein (1990) shows that one can have high

Table 4 God classes for participants working on the Solitaire-Freecell program (task carried out without visualization)

	Solitaire-Freecell (23 classes)				
God Class	**Participant**				
	F13	**F15**	**F25**	**F31**	**F41**
ControladorGlobal	Yes		Yes	Maybe	Yes
FrameFreecell	Maybe	Yes	Yes		Maybe
InterfacePaciencia		Maybe	Maybe		Maybe
Estoque		Maybe			

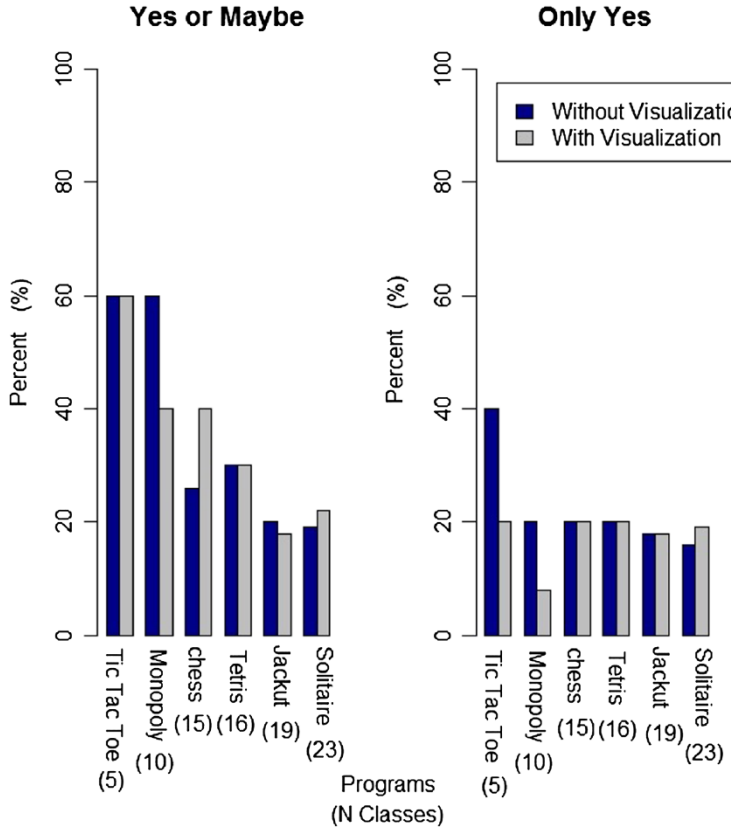

Figure 3 Percentage of god class candidates.

agreement rate and low values of the Kappa coefficient, when the variance on values of raters is low. We noted this situation in the work of Schumacher et al. The Finn coefficient is recommended when variance between raters is low (Finn 1970). Whitehurst (1984) proposes Finn as an alternative to problems with Kappa, and affirms that it is the most reasonable index for agreement.

To make the comparison of agreement values for the cases with and without visualization easier, we adopted classification levels. We used the same defined by Landis and Koch (1977), such as Schumacher et al. (2010) had done. Landis and Koch proposed the following classification: slight, for values between 0.00 and 0.20; fair (between 0.21 and 0.40); moderate (between 0.41 and 0.60); substantial (between 0.61 and 0.80); and almost perfect (between 0.81 and 1.00) agreement.

Table 5 presents Finn values for the programs considering "yes" or "maybe" options. On the left, we present values for the case of without visualization. On the right, we present values for the case with visualization. There were two cases where the agreement level was higher with visualization: Monopoly (from moderate to substantial) and Jackut (from substantial to almost perfect). There was a reduction in agreement for Tic Tac Toe (from moderate to slight), but it is the only change without significance (p-value = 0.283), therefore we did not consider it in the agreement analysis. Table 6 considers only cases where participants were sure about the god classes. There were two cases where the level of agreement was higher with visualization (Tic Tac Toe and Chess, changing from

Table 5 Agreement among participants, considering "yes" or "maybe" marked

Program	Without visualization					With visualization				
	Number of classes	Raters (participants)	Finn coefficient	p-value	Agreement	Number of classes	Raters (participants)	Finn coefficient	p-value	Agreement
Monopoly	10	6	0.507	0.000996	Moderate	10	4	0.7	7.37e-05	Substantial
Tetris	16	6	0.733	5.09e-12	Substantial	16	5	0.8	4.91e-13	Substantial
Tic Tac Toe	5	6	0.6	0.00335	Moderate	5	5	0.2	0.283	Slight
Chess	15	5	0.787	1.21e-11	Substantial	15	6	0.664	4.46e-08	Substantial
Jackut	19	4	0.772	1.73e-10	Substantial	19	6	0.881	1.33e-27	Almost perpect
Solitaire	23	5	0.843	4.38e-22	Almost perfect	23	5	0.843	4.38e-22	Almost perpect

Table 6 Agreement among participants, considering only "yes" marked

Program	Without visualization					With visualization				
	Number of classes	Raters (participants)	Finn coefficient	p-value	Agreement	Number of classes	Raters (participants)	Finn coefficient	p-value	Agreement
Monopoly	10	6	0.827	8.44e-12	Almost perpect	10	4	0.867	3.87e-09	Almost perpect
Tetris	16	6	0.892	5.34e-25	Almost perpect	16	5	0.675	5.89e-08	Substantial
Tic Tac Toe	5	6	0.653	0.00102	Substantial	5	5	0.84	7.14e-06	Almost perpect
Chess	15	5	0.787	1.21e-11	Substantial	15	6	0.84	5.26e-18	Almost perpect
Jackut	19	4	0.842	3.31e-14	Almost perfect	19	6	0.895	6.683-30	Almost perpect
Solitaire	23	5	0.896	3.62e-29	Almost perfect	23	5	0.843	4.38e-22	Almost perpect

substantial to almost perfect agreement). There was one case where the level of agreement was lower with visualization (Tetris changing from almost perfect to substantial).

RQ2: How well do humans and an oracle agree on identifying god classes?

To deepen the analysis of the impact of conceptualization on god class identification, we extended the human performance questions in Schumacher et al. (2010) using an oracle and comparing the answers of the oracle and participants. The oracle was made up of two experienced researchers in academia and industry. Each of the researchers did the task independently and without any contact with the participants' answers. We show their answers in Table 7.

We used the Finn coefficient (1970), the same as the previous RQ, to test the agreement. Table 8 shows the results. Tic Tac Toe, Tetris and Solitaire all had an agreement. The Chess and Jackut programs had one disagreement. The Monopoly program had two disagreements.

After these observations, the researchers met to discuss the differences and to define the oracle. An interesting observation is that the researchers noted that they were very strict in their analysis. Due to this, for some cases (two classes for Monopoly, one class for Chess, one class for Solitaire and one class for Jackut), classes were deleted from the list. We highlight the class FrameFreeFreecell in the Solitaire program. In this case, both researchers found that the class was a candidate to be god class. However, during the meeting, the researchers were more flexible about the size and the few methods out of scope, because the class represents the graphical user interface of the program. After the meeting, the oracles reached the agreement presented in Table 9.

Figure 4 shows the distribution of the Finn coefficient, comparing the agreement among the participants and the oracle. Let us initially focus on Figure 4(A) and (B), confirmed and possible god classes. It is possible to note that the average agreement with visualization

Table 7 Oracle answers

Program	God class	Oracle	
		Or1	Or2
Ti Tac Toe (5 classes)			
	-	-	-
Monopoly (10 classes)			
	Jogo	Yes	Yes
	Tabuleiro	Maybe	Yes
	UserStoriesFacade	-	Maybe
	Jogador	-	Maybe
Chess (15 classes)			
	Chess	Yes	Yes
	BoardGUI	-	Yes
Tetris (16 classes)			
	Tetris	Maybe	Maybe
Jackut (19 classes)			
	Usuario	-	Maybe
Solitaire (23 classes)			
	FrameFreeFreecell	Maybe	Yes

Table 8 Finn coefficient among the researches considering "yes" marked and "yes" or "maybe"

			Finn coefficiente	
Program	**Subjects (Nclasses)**	**Raters (oracle)**	**Yes or maybe**	**Only yes**
Ti Tac Toe (5 classes)	5	2	1	1
Monopoly	10	2	0.6	0.8
Chess	15	2	0.867	0.867
Tetris	16	2	1	1
Jakut	19	2	0.895	1
Solitaire	23	2	1	0.913

is higher only for the Monoploy program, is the same for Tic Tac Toe, and is lower for the other four cases. Figure 4(C) and (D) focus on confirmed god classes. First, one should notice that, as expected, the average is higher and the variances are smaller than in Figure 4(A) and (B). Comparing Figure 4(C) and (D), the values are higher for cases with visualization for three cases (Chess, Solitaire and Tic Tac Toe), and lower for the other three cases (Jackut, Monopoly and Tetris).

RQ3: Which strategies are used to identify god classes?

We used the logged actions to investigate the strategy adopted by the participants to detect god classes. We analyzed two aspects. The first one was the differences between relevant actions for the cases, with and without visualization. The second were the preferences of the participants. We grouped participants with similar preferences and evaluated if there was a "better" strategy.

Relevant actions

Table 10 shows the number of actions performed during the experiment by all participants. We grouped the columns "what", "kind" and "description" from the log and shortened the terms presented in Figure 2. The first action represents the activation of the Package Explorer view. The Package Explorer shows the Java element hierarchy of Java projects. It is a tree view that shows Eclipse projects on the first level, folders on the

Table 9 Final oracle answers

Program	**God class**	**Oracle**
Ti Tac Toe (5 classes)		
	-	-
Monopoly (10 classes)		
	Jogo	Yes
	Tabuleiro	Maybe
Chess (15 classes)		
	Chess	Yes
Tetris (16 classes)		
	Tetris	Maybe
Jackut (19 classes)		
	-	-
Solitaire (23 classes)		
	-	-

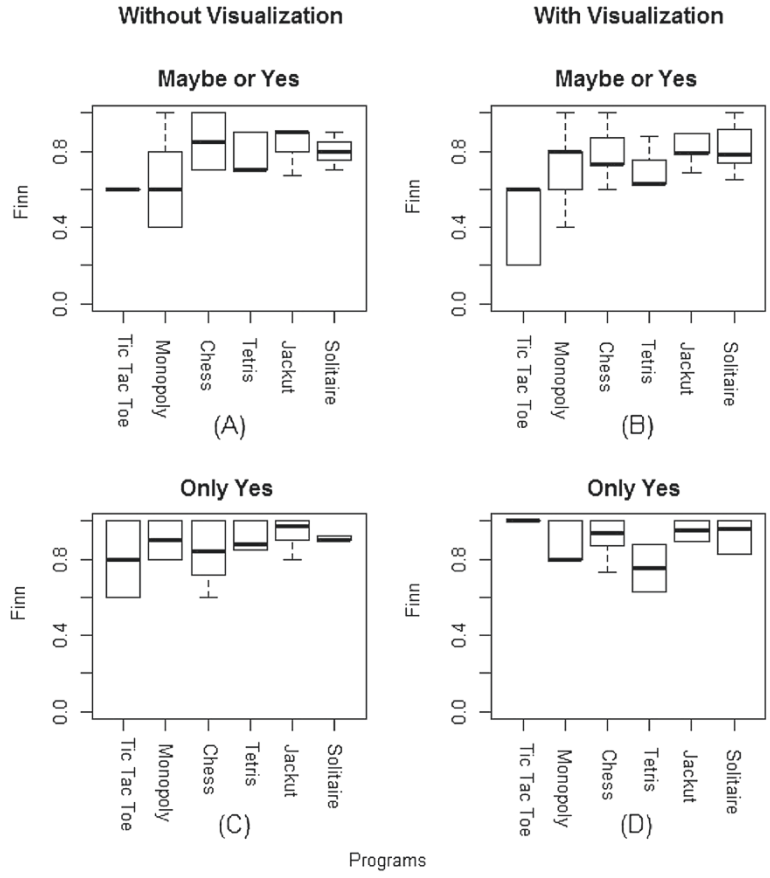

Figure 4 The distribution of Finn agreement values between participants and oracle: cases without visualization in (A) and (C); cases with visualization in (B) and (D).

second level, packages on the third level, classes on the fourth level, and methods and attributes on the fifth level. It is used to navigate in the original Eclipse's set up.

The second relevant action represents the activation of the Compilation Unit Editor, which is commonly used for code reading/writing. The actions numbered three to seven represent the activation of views of the visualization tool. For example, action three represents the activation of the Polimetric view, presented in Section 'Tools'. Actions four

Table 10 Actions performed with and without the visualization tool

No	Actions	Visualization	
		With	**Without**
1	activated_view_PackageExplorer	311	806
2	activated_editor_CompilationUniEditor	263	1240
3	activated_view_PolymetricView	135	
4	activated_view_DependencyView	117	
5	activated_view_TreeMapView	114	
6	activated_view_CouplingMatrixView	70	
7	activated_view_GridMatrixView	59	
8	Others: 10 with SourceMiner and 23 without SourceMiner	47	71
	Total	1116	2117

to seven represent the activation of the Dependency, Tree Map, Coupling Matrix and Grid Coupling views. The other actions were not shown because they did not occur very often.

It is possible to note the difference in the total. For the case with visualization, the total number of actions was 1116. Without visualization the total number was 2117. The views of the visualization tool were used frequently, when permitted: Polymetric (135), Dependency (117), Tree Map (114), Coupling Matrix (70) and Grid Coupling (59).

Strategies of god class detection

To evaluate the strategies we only observed cases with visualization because, in the case without visualization, the main action was related to activation of the Compilation Unit Editor, i.e. reading source code, and Eclipse does not log its use in detail.

As discussed in Section 'Data', we defined a sequence as all actions performed by the participant for a program.

In this experiment, there were three sequences of actions for each participant using the visualization tool, one for each program. There were 11 participants, but we deleted three of the sequences because we found problems in these sequences. The problems were caused by misuse of the Task Register plug-in. Therefore, 30 sequences were evaluated.

Our first analysis was related to individual sequences. We searched for common patterns. To do this, we used the LTL Checker of the ProM[b], a support tool for techniques of process mining (van der Aalst 2011). With the LTL Checker it is possible to check a property of the set of sequences expressed in terms of Linear Temporal Logic (LTL).

Consider the following three sequences, as a simple example. The sequences one and two have three actions, and the third sequence has two actions:

1. *activate_view_Polimetric, activate_view_TreeMap, open_CompilationUnitEditor*
2. *activate_view_Polimetric, open_CompilationUnitEditor, activate_view_TreeMap*
3. *activate_view_Polimetric, activate_view_TreeMap*

The LTL Checker allows checking, for example, that the action *activate_view_ TreeMap* always occurs some time after *activate_view_Polimetric*. It is also possible to check that the action *open_CompilationUnitEditor* occurs in two out three sequences. Or that the action *activate_view_TreeMap* occurs next to *activate_view_Polimetric* in two out three sequences. Table 11 shows the main results and interpretations for the sequences of the experiment.

We also investigated preferences grouped by participants. Some participants read more and used fewer views, whereas others did the opposite. To evaluate these aspects, we calculated the ratio between the number of classes investigated for each program and the use of views and readings. We counted the number of actions related to reading (activation of Compilation Unit Editor), activation of hierarchical views (Polymetric or Tree Map), and activation of coupling views (Dependency, GridCoupling or CouplingMatrix). For example, there were 12, two and four (18 in total) "activated_editor_CompilationUnitEditor" for the sequences of the F13 participant, for the programs Monopoly, Tetris and Tic Tac Toe, respectively. The total number of classes for these three programs is 31. We defined the ratio of using the CompilationUnitEditor, for F13 participant, as 18/31. He/she activated Coupling views 31 times for the three programs. In this case, the ratio is 31/31. Note that, if the participant activated the views more than the total number of classes of the

Table 11 Main results of the evaluation of sequences with LTL checker

Formula eventually_activity...	Action (activity) Activitation of ...	Number of cases (in 30)	Interpretation
A	Compilation Unit Editor	20	There were 20 cases where the participants read source code to identify god classes
A, B, C, D and E	Polimetric, TreeMap, Grid Coupling, Coupling Matrix, Dependency	11	Despite hierarchical views ad coupling views show the same attributes, there were 11 cases where the participants adopted all views of the visualization tool to identity god classes
A and B	Hierarchical views and Coupling views	27	There were 27 cases where the participants combine the use, at least, one of the hierarchical views and one of the coupling views to identify god classes
A, B and C	Hierarchical views and Coupling views and Compilation Unit Editor	17	There were 17 cases where the participants combined reading source code with one of the hierarchical (at least) and one of the coupling views (at least)

investigated programs, the ratio will be greater than 1. The complete results are presented in Figure 5.

From the figure, we identified six different profiles. We present them in Table 12. The first profile is composed of participants F14, F21 and F42. They used very little or no reading, and had a slight preference for coupling views. In profile 2, the two participants (F31 and F41) also did little reading in comparison to the usage of coupling views. In profile 3, participants F15 and F25 had preferences for coupling views, but they focused on reading unlike the previous group. For the other cases we found only one participant.

To investigate the "quality" of profiles we evaluated the agreement between each participant and the oracle. We plotted the Finn coefficient in Figure 6. From the graphs, it can be seen that no participants strongly agreed with the oracle. For example, participant F21 (Figure 6(A) and (C)) had the highest agreement for the program Chess and the worst agreement for Jackut. This was the general pattern.

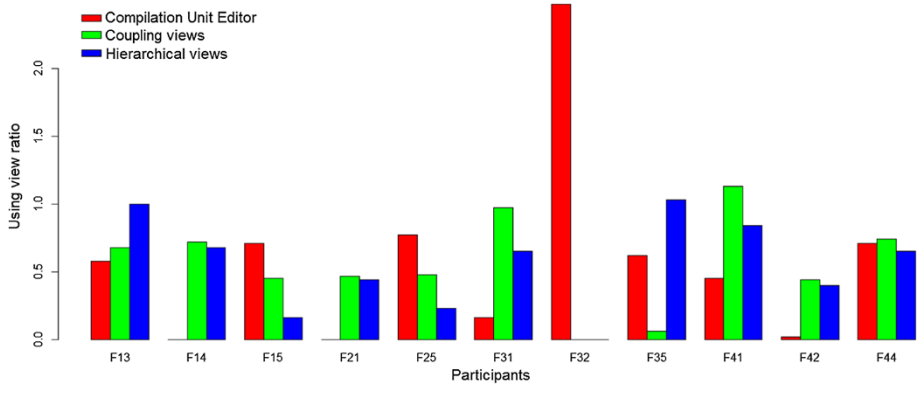

Figure 5 Ratio of using CompilationUnitEditor, hierarchical and coupling views.

Table 12 Profiles of participants related to the use of reading, hierarchical and coupling views

No	Profile	Participants
1	No reading and slight preference by coupling views	F14, F12 and F42
2	Few reading and strong preference by coupling views	F31 and F41
3	Focus on reading and preference by coupling views	F15 and F25
4	Preference by hierarchical views	F35
5	Slight preference by hierarchical views	F13
6	Similar using of all views and reading	F44
7	None using the visualization tool	F32

RQ4: What issues in code lead humans to identify a class as a god class?

To address this question we collected data related to drivers used by participants on the identification of god classes. The answer form provided a multiple choice list of nine drivers extracted from the work of Schumacher et al. (2010) (see Section 'Forms'). It also allowed the participants to write down new drivers.

For cases where visualization was not used, the participants wrote down 19 short descriptions of new drivers. The coding process on these 19 new drivers' descriptions was simple. For example, some participants wrote down "class has many lines of code", others wrote down "class has big size". In these cases, we defined "class has high LOC" as the driver. After this process, we narrowed the descriptions down to six actually new drivers. Table 13 shows all the drivers. The most common drivers were "class is highly complex" (48 times) and "method is highly complex" (31 times). An intermediate group included drivers like "class is special/framework" (12 times), "class represents a global function" (8 times) and "lack of comments" (7 times). The other drivers had low marks (≤ 5 times).

We also analyzed the distribution of drivers by participants. The drivers "class is highly complex" and "method is highly complex" were filled by all and almost all participants, respectively. The other 13 drivers were used by no more than four participants,

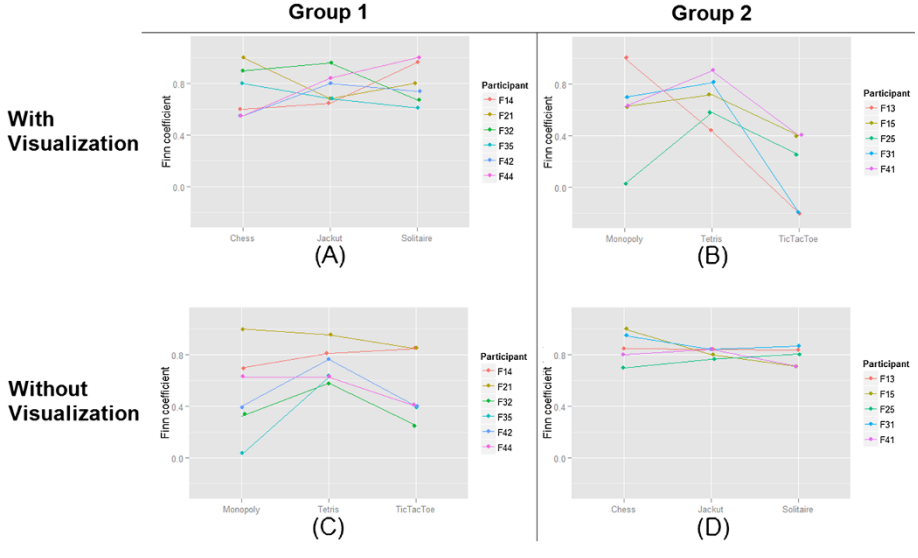

Figure 6 The Finn agreement coefficient among participants and the oracle: Group 1 in (A) and (C); and Group 2 in (B) and (D).

Table 13 Drivers by participants (Schumacher et al. and new drivers) without visualization

Participant	Based on Schumacher *et al.* coding									Coding on comments					
	Class				Method			Attribute	Method/class	Class				Method	
	Not used	Highly complex	Misplaced	Special/ framework	Wrong named	High complex	Misplaced	Not used	Lacks comments	High LOC	Many dependencies	Many methods	Global functions	Can be split	High LOC
F13	0	6	0	3	0	6	2	0	3	0	0	0	0	1	1
F14	0	3	0	0	1	1	1	0	0	1	0	0	0	0	0
F15	0	4	0	0	1	3	0	0	0	0	0	0	0	0	0
F21	0	4	0	4	0	0	0	1	0	0	0	0	0	0	0
F25	0	9	0	0	0	6	0	3	1	0	0	0	0	0	0
F31	0	1	0	0	0	1	0	0	0	0	0	0	1	0	0
F32	0	4	0	0	0	0	0	0	0	0	0	0	7	0	0
F35	0	5	0	0	0	2	1	0	0	0	0	0	0	0	0
F41	0	5	0	2	0	7	0	1	0	3	0	2	0	0	2
F42	0	3	0	3	0	1	0	0	0	0	4	0	0	0	0
F44	0	4	1	0	0	4	0	0	3	0	0	0	0	0	0
Totais	0	48	1	12	2	31	4	5	7	4	4	2	8	1	3

even the drivers in the intermediate group. This indicates that drivers like "class is special/framework" were consistently adopted by some, but this was not a consensus among the participants.

For cases where visualization was used, the participants wrote down 27 short descriptions of "new drivers". Like before, these descriptions were reduced to a group of six actually new drivers. This group was almost the same as before. As shown in Table 14, it excludes/includes just one driver ("method can be split" is substituted by "problem with hierarchy"). The most common drivers were "class is highly complex" (46 times), "method is highly complex" (21 times) and "class has many dependencies" (20 times). The intermediate drivers were "class is special/framework" (11 times), "class/method lack comments" (8 times) and "class has high LOC" (7 times). The others had low marks (≤ 3 times).

Like before, "class is highly complex" and "method is highly complex" were chosen by all or almost all participants. "class has many dependencies" was also chosen by most (seven) participants. "class is special/framework" was chosen by five. The other 11 drivers were chosen by less than three participants, following the same pattern previously described.

Finally, we analyzed the differences by cases with and without visualization. The coding we performed produced practically the same new drivers and the distribution of the total marks was also quite similar. The most common drivers were the same: "class is highly complex" and "method is highly complex". The notable difference was the "class has many dependencies" driver. It was common in the visualization-based analysis and uncommon in the code-based analysis (without visualization).

Discussion

This section presents the discussion of the results of the experiment. Following the same logic of the previous section, we created one subsection to present the discussion for each question.

RQ1: How well do the participants agree on identifying god classes?

We addressed this question from two perspectives: i) number of candidate classes and ii) agreement test.

Number of candidate god classes

Comparison with Schumacher et al. (2010)'s work. Schumacher et al. (2010) found a very small number of god class candidates: only two in 52 inspected for one of the projects and three in 51 for the other project (3.8% and 5.8%, respectively). The numbers were much higher in our case: the average number of candidate classes were 34.8% and 35.8% for cases with and without visualization, respectively. One possibility is that the difference was related to the type of programs. Because the programs used in our experiment have a smaller number of classes, one god class candidate represents a high percentage.

On the other hand, it is possible to conjecture about the difference on the number and experience of the participants. In both studies, the number of participants is small: 11 in our case, and only four in the Schumacher et al. work. Our participants were undergraduates and Schumacher et al. ran the study with professionals. However, despite experience, three of their participants were unfamiliar with the concept of smell and god class in the experiment. The other participant had only heard about god classes before. Therefore, for both studies the participants had little knowledge about the concepts. We consider this

Table 14 Drivers by participants (Schumacher et al. and new drivers) with visualization

Subject	Based on Schumacher *et al.* coding									Coding on comments					
	Class				Method			Attribute	Method/class	Class					Method
	Not used	Highly complex	Misplaced	Special/ framework	Wrong named	High complex	Misplaced	Not used	Lacks comments	High LOC	Many dependencies	Many methods	Global functions	Hierarchy structure	High LOC
F13	0	4	0	1	0	4	0	0	2	3	4	1	0	0	1
F14	0	5	0	0	1	1	0	0	0	1	3	0	0	0	1
F15	0	3	0	0	0	1	0	0	0	0	0	0	0	0	0
F21	0	4	0	5	0	0	0	1	0	0	0	0	0	0	0
F25	0	8	0	0	0	3	0	1	2	0	0	0	0	0	0
F31	0	1	0	0	0	1	0	0	0	0	1	0	0	0	0
F32	0	2	0	3	0	0	0	0	0	0	0	0	2	0	0
F35	0	6	0	0	0	3	2	0	0	0	0	0	0	0	0
F41	0	3	0	1	0	3	0	1	0	3	2	0	0	1	0
F42	0	6	0	1	0	3	0	0	0	0	5	0	0	0	0
F44	0	4	0	0	0	2	0	0	4	0	1	0	0	0	0
Totais	0	46	0	11	1	21	2	3	8	7	20	1	2	1	2

aspect empirical evidence that the knowledge related to reading or having heard about the concept of smell or god class is weaker than the experience related to work in a development environment. We usually believe in that, but we do not have much empirical evidence. To us, this makes the importance of expanding the discussion on the context of software engineering experiments evident. This problem has been rarely addressed in Experimental Software Engineering (Dybå et al. 2012; Höst et al. 2000, 2005).

Comparison between cases with and without visualization. There was no significant difference between the number of god class candidates with and without visualization (Figure 3). Considering the "yes" or "maybe" marks, values for cases with visualization are higher for the two programs, and lower for two other programs. Considering only "yes" marks, values for cases with visualization are higher only for one program and lower for two programs. The main finding is that the use of visualization does not impact the number of candidate god classes, i.e. people do not identify more or fewer god classes because of a better comprehension of the design of the program (obtained via the visualization tool). For us, this is evidence that people have their own conceptualization of what a god class is, and the visualization tool does not affect this.

This finding appears to run contrary to the one presented by Murphy et al. 2010. In this work, the first hypothesis is "Programmers identify more smells using the tool than not using the tool". However, their tool uses visual aids based on detection mechanisms. These mechanisms use metrics to visually highlight certain smells related to attributes. This contributes to harmonize (for better or worse) the conceptualization of what a smell is.

Agreement test

Comparison with Schumacher et al. (2010)'s work. We could not compare both works in terms of numbers because Schumacher et al. used Cohen's Kappa and we adopted the Finn coefficient. In Shcumacher's et al. work, the agreement for the one project was -2%. Based on Landis's and Koch's (1977) interpretation of Kappa, this indicates no agreement among the two participants. In the other project, the calculated Kappa was 48% (suggesting moderate agreement). Their conclusion was that there was not a high level of agreement. The work of Mäntylä (2005), in a similar study about agreement, found a similar result. For comparative purposes, we tested agreement with the Fleiss Kappa coefficient[c] and Landis's and Koch's interpretation. Our results showed only a slightly higher agreement than Schumacher's et al. work.

The values of the Finn coefficient were significantly higher as we can see in Tables 5 and 6. However, it is important to note that the Finn and Kappa tests measure agreement using both god and non-god classes. Consequently, we were cautious about finding high levels of agreement (almost perfect), because many classes are clearly not god classes. Participants were expected to agree on this. The same situation might have happened in the Scumacher's et al. work. We believe that their values were affected by the problem of the Kappa coefficient, which we discussed in the Subsection 'Finn agreement test'. Despite this, the results considering just the "yes" mark were convincing. Almost perfect agreement occurred for four of the six cases without visualization, and five of the six cases with visualization. The results were not as convincing when some level of doubt was allowed. For the "maybe" and "yes" marks, almost perfect agreement occurred in only

one of the six cases without visualization, and two of the six with visualization. Based on these analyses, we consider our results aligned with previous studies: we did not find a high level of agreement.

The main finding here is that the participants have different conceptualizations of what makes a god class. This may happen in two dimensions. First, participants may have different classification thresholds in their evaluation of candidates. For example, participants may have different perceptions of how many roles a class can assume before it becomes a god class. Another possibility is that participants simply have different views of the god class concept. We consider this to be the weaker possibility because they had the same training and they had a similar level of knowledge before the experiment. Furthermore, the difference in the classification threshold was observed first hand when the oracle was being defined by more experienced professionals.

Comparison between cases with and without visualization. Another important dimension is the impact of visualization. We can not affirm that the use of visualization leads to "big" improvements. We can see in Tables 5 and 6 that there were four in 11 cases where the level of agreement was higher and there was one case where the level of agreement was lower with visualization. As a result, the agreement was slightly better for cases with visualization. We use the next research question to further investigate the impact of visualization.

RQ2: How well do humans and an oracle agree on identifying god classes?

We addressed this question from two perspectives: i) the oracle definition process; and ii) the agreement between the oracle and participants.

The oracle definition process

An important aspect in the agreement analysis between participants and the oracle is that we can not claim that the discrepancies between the participants and the oracle are errors, but we can claim that the oracle definition process was more rigorous. During the oracle meeting, both oracle researchers noted that the choice of a god class was associated with their personal perception about "how many roles the class needs to be considered god class" or "how much LOC the class needs to be considered a large class", or other subjective thresholds related to the characteristics of classes. This can be an explanation for the lack of agreement found among the participants of our study, as well as in (Mäntylä's 2006a and Schumacher's et al. 2010) studies.

The agreement between the oracle and participants

We analyzed this from two perspectives: level of certainty or doubts ("yes" or "maybe" marks) , and the use of visualization (Figure 4).

Comparison between cases with and without doubts. From Figure 4, we can see that when certainty – "yes" mark – is involved, the agreement between the participants and the oracle is generally high, while the variation in these agreements – see the box sizes – is usually small. These values are significantly reduced for the "yes or maybe" marks. The agreement average and variation between the oracle and the participants are much lower. If we consider the discussion on the high level values naturally produced by the Finn

coefficient, this reinforces the thesis that agreement is low when doubt is involved. This also reinforces the evidence that participants have a personal conceptualization of what is a god class. As discussed previously, the likely causes for this is the different classification thresholds, or the different views of the god class concept. Like before, we consider the former more probable.

Comparison between cases with and without visualization. Considering the averages and the "yes or maybe" options in Figure 4(A) and (B), one can observe that there was only one case where the visualization improved agreement. Considering only the "yes" option (Figure 4(C) and (D)), there were three out of six cases where the visualization improved agreement. We conclude that the visualization does not affect the agreement values. This weakens the argument made in the discussion of RQ2, where we stated that visualization slightly improves the agreement among the participants. It might be the case that visualization slightly improves the precision among the participants, but it does not improve the accuracy of their detection against an oracle reference.

RQ3: Which strategies are used to identify god classes?

Relevant actions

First we addressed differences in the amount of reading for the cases with and without visualization. Comparing both cases, the reduction in the activation of the Compilation Unit Editor, or the reading of source code, was significant: from 1240 (without visualization) to 263 (with visualization). On the other hand, actions related to the activation of the views of the visualization tool emerged. We concluded that participants exchanged reading for the observation of views, i.e., they used visualization to search for attributes that indicate god classes. Based on the use of the views and tacit knowledge, we suggest that, to identify god classes, participants compared LOC, complexity or number of methods; observed coupling attributes and read code to understand the context of the class in the program.

Strategies of god class detection

Evaluating sequences. We analyzed each case presented in Section 'Strategies of god class detection'. The first case was related to reading. It was found in 20 of the 30 sequences. We believe that it was adopted by the participants to comprehend the role of the class in the program. Considering the concepts presented by (Fowler 1999 and Lanza and Marinescu 2005), some reading is necessary. In the definition of a Large Class, Fowler says that "*...it often shows up as too many instance variables*". In the God Class and Brain Class definitions, Lanza and Marinescu adopt metrics that involve a number of branches, deep nesting and a number of variables. In the experiment, it was not possible to see these characteristics with the visualization tool. However, despite the expected behavior, 10 sequences did not contain actions related to reading code. We analyze these aspects in the evaluation of participants hereafter.

In the other formula, we detected that in 11 sequences participants activated all five views of the visualization tool. As discussed in Section 'Tools', the two hierarchical views present similar attributes, like the other three coupling views. This indicates that in some cases participants worked more than necessary. Our conjecture is that they wanted to increase their level of certainty. It is important to note that, for most sequences (19), the

participants did not use all the views, which we consider more efficient. Despite this, the need to confirm ideas is a behavior that must be considered in the analysis.

Another analysis that we performed was related to the combination of views and reading. The combination of hierarchical and coupling views occurred in 27 of the 30 sequences. This was the main strategy used to detect god classes.

Evaluating participants' strategies. We identified three participants who combined structural, coupling views, and used very little or no reading. It is profile 1 in Table 12. In profile 2, the two participants also used little reading in comparison with their preference for coupling views. From these two profiles, we suggest that the preference for coupling indicates that this attribute is the strongest, and that reading source code was not considered so relevant in god class detection. Another interesting profile is the number 3. In this case, the participants focused on reading and had a preference for coupling. In comparison with the other two profiles, this agrees with our previous discussion that participants exchanged reading for observation of specific attributes in the views of the visualization tool. In some cases, participants seem to be more confident reading than using the visualization tool. The other profiles indicate different preferences. For example, profile 4 shows a strong preference for structural views and profile 6 shows the similar use of all types of views and reading.

An interesting observation is related to the absence of a "better" strategy, or even a "better" participant, considering the agreement with the oracle (Figure 6). This reinforces our idea that god class detection is strongly affected by personal conceptualizations. If this was not true, a participant with a similar approach to the oracle would produce high levels of agreement with it for most programs. This was not what happened in our experiment.

RQ4: What issues in code lead humans to identify a class as a god class?

Some drivers were chosen by participants, independent of visualization. "Class is highly complex" (chosen by all participants) and "Method is highly complex" (chosen by nine out eleven participants), two of the strongest drivers, are examples of that. The only, and very interesting exception to the rule, is the "class has many dependencies" driver. Only one participant chose this driver for the case without visualization, and seven participants chose the driver for the case with visualization. While the visualization tool provides several views which showed signs of dependency, the current IDEs and the source code do not help in the identification of dependency. This case is evidence that the use of a visualization tool can indeed help, because some detection drivers are poorly supported by the current state of the practice. This, however, does not solve the conceptualization problem.

It is also worthwhile to observe that, although participants did not always use the same drivers, they were very consistent in using their drivers of choice. For "Method is highly complex", for example, the two participants who did not choose the driver are the same (F21 and F32) with and without visualization. Other cases where the same participants chose the same drivers were: "attribute is not used", "methods or class lack comments". For other cases, the difference between the choices of the participants is very small; at most one participant. That suggests that in some cases the choice of drivers is a personal issue. This idea reinforces our conjecture about the importance of the community discussing the problem of conceptualization on smells in depth.

Summary of findings and insights

In this work, we addressed the problem of conceptualization in the god class detection, from different perspectives. In this section, we gather the findings and present our conclusions. We also present peripheral, but not less important findings:

- Agreement
  - *Related to the conceptualization problem*: The low agreement rate showed us that the problem exists. However, we consider the question related to visualization more interesting, because it showed us where the problem is, or at least where it is not. The problem of conceptualization is that it is not related to the comprehension of design, as we had expected. We argue this because visualization did not increase agreement. The finding is that the problem is related to personal understanding of the smell concepts or of personal thresholds adopted.
  - *Peripheral findings*: We found evidence that experience affects the degree of agreement. Our evidence was: i) the comparison of the number of candidate god classes in our and Shumacher's et al. work; and ii) low agreement among the participants and our oracle (made up of more experienced researchers). Another interesting peripheral finding is related to the using of the Kappa coefficient for agreement analysis. We noted that, for the Kappa coefficient, in some cases, the agreement value is high and the coefficient is low. Based on our research, we suggest the adoption of the Finn coefficient (see discussion in the Section 'Finn agreement test').
- Strategies
  - *Related to conceptualization problem*: The absence of a "better" strategy reinforced our finding that each participant has his/her own idea about how and what he/she has to do to identify god classes.
  - *Peripheral findings*: We proposed an approach to identify strategies: the identification of used views from the logs. We grouped participants according to their strategies. The main profile identified focused on coupling attributes, and little reading.
- Decision drivers
  - *Related to the conceptualization problem*: As mentioned under "strategies of god class detection", some decision drivers are also personal choices. In a consistent way, participants had different preferences for characteristics they used to identify god classes. This also reinforced our previous findings.
  - *Peripheral findings*: The main decision driver was "class is highly complex".

The results discussed here strengthen the idea that the problem in god class detection is more related to conceptualization than to making the comprehension of the code design easier by facilitating the (visual) observation of certain attributes of the code. Note that this type of problem is not solved by using other observation approaches, such as proposing a new metric. The key problem is to identify concrete "good" examples of smells, providing standard definitions of them, teaching people about their conceptualization, and

only then providing the tools and methods (using metrics, thresholds or visualizations) to aid their identification.

Threats to validity

Our analysis of threats was based on Wohlin et al. (2012).

External validity. Our first threat fits in the "interaction of selection and treatment" subcategory and is related to the fact that the participants in our experiment were undergraduates and had little experience in a real software development environment. Moreover, the experiment was run with 11 participants. Although the aspects related to the participants could be considered a problem for generalization, we have strong evidence that the existence of the problem discussed in this work is intrinsic to smell detection and also happens with experienced developers. The evidence is: i) smell concepts are presented subjectively Fowler (1999) or are dependent on thresholds Lanza et al. (2005); ii) findings show low agreement in other works with more experienced participants Schumacher et al. (2010); and iii) the initial lack of agreement among the experienced researchers who created the oracle in this work. We believe that experience affects our results only in terms of intensity, but the problem of conceptualization exists for all cases. We are planning to replicate this experiment with more experienced participants to evaluate the impact of experience.

Other threats to external validity fit in the "interaction of setting and treatment" subcategory. In this case, the threat is the type of program. We adopted simple programs. Another point is the domain: they are familiar software, and in most of the cases, games. Software of a different domain might present different characteristics. However, we argue that the same, previous reasoning is valid here, weakening the threat: the phenomenon can be studied in any type of software. In fact, because we used simple and familiar software, the problem should be minimized, which was not the case. Moreover, we did not address the difficulty of god class identification, but how conceptualization affects smell detection.

Internal validity. The study has threats in two subcategories related to internal validity. The first one is "ambiguity about direction of causal influence". We highlight the fact that the training about god classes reflects the view of the experimenter. The view of the experimenter could affect the participants' conceptualization and their ideas about what they had to look for to identify a god class. To minimize this effect, we limited the time of the training in the god class concept and adopted the support questions from Schumacher's experiment to guide participants in god class detection during our experiment. These actions also mitigate the same threat in the opposite way: participants could not have a general idea of what to search for. We consider that the training and the support questions steered the participants to search for classes that represent the god class concept adopted.

Another threat is the training in the visualization tool. In our feedback form, participants indicated that the quality of training was good, in general. However, we can not confirm that it was sufficient to prepare participants in the use of visualization. Another subcategory of the internal validity is "maturation". Participants could be affected because they do the same task over six programs, so they may learn as they go and work faster. On the other hand, they could be affected negatively because of boredom. We consider maturation a weak threat because the experiment was performed in 1.5 hours, on average. We consider this a reasonable period of time to do a task in a balanced way.

Conclusion validity. In the "reliability of measures" category, we should report that the logged information represents the actions of the participants only indirectly. They actions of the Eclipse IDE. For example, if a developer changes the perspective in the Eclipse, some views are activated by the tool and these actions are registered in the log. To mitigate this aspect, we investigated the logging to evaluate actions in detail and eliminated lines clearly related to Eclipse actions. Moreover, these registers occurred for all participants and did not affect the general conclusion. In the "reliability of treatment implementation", we have to consider that a participant who could have used visualization may have completely disregarded the views of the visualization tool. However, we checked the UDC logging and only one participant did not use the visualization resources. Because this occurred in one case, the results were not affected. Lastly, due to the number of data points, some of our findings were based on the analysis of graphs and tables, and inferential testing was done in a few cases.

Context and related works

As the use of the concept of smells has become widespread, empirical studies have been presented to help understand their effects. As discussed in Section 'Background', we identified three types of empirical work in the area. In this section we present them. We provide more details of papers related to human aspects, which are closer to our work. However, we present correlation studies because they provide evidence that the use of code smells as an indicator of a problem in the design has been inconsistent, which we believe might be caused by the problem of conceptualization. We also present some works related to the use of support tools, because our work is focused on the use of a visualization tool.

Correlation studies

This type of work usually focuses on analyzing software evolution and tries to link smells with some characteristic of code. Normally, they investigate data in software repositories. Olbrich et al. (2010), for example, investigated the influence of two smells ("God Class" and "Brain Class") on the frequency of defects. In order to do this, they analyzed historical data from three open-source software systems. They found that, in specific cases, the presence of these smells might be beneficial to a software system. In another study, Olbrich et al. (2009) investigated the evolution of two other smells ("God Class" and "Shotgun Surgery") for these same systems. Li and Shatnawi (2007) studied the relationship between smells and error probability. They investigated three error-severity levels in an industrial-strength open source system. Their findings indicate that some bad smells are positively associated with the probability of errors.

Sjoberg et al. (2013) presented a controlled study where six professionals were hired to maintain four systems for 14 days. Their aim was to quantify the relationship between code smells and maintenance effort. One of the main findings was that "the ... smells appear to be superfluous for explaining maintenance effort". Abbes et al. (2011) adopted the concept of anti-pattern, which, like the concept of code smell, presents "poor" solutions to recurring design problems. They performed an empirical study to investigate whether the occurrence of anti-patterns does indeed affect the understandability of systems by developers during comprehension and maintenance tasks. They found that the

occurrence of one anti-pattern does not significantly decrease developers' performance while the combination of two anti-patterns significantly impedes developers.

Smells and support tools

Some works evaluated smell detection using automatic detection tools (Moha et al. 2010; Mäntylä and Lassenius 2006a; Schumacher et al. 2010). Other works addressed smell detection with visualization tools (Carneiro et al. 2010; Murphy-Hill and Black 2010; Parnin et al. 2008; Simon et al. 2001). From the former, we discuss the first tool, because it was developed to mitigate subjectiveness. The authors propose DECOR. It is a method/tool "that embodies and defines all the steps necessary for the specification and detection of code and design smells". Despite the tool mitigating some subjectiveness, the focus of the work was on the method, not on the aspect that affects (or does not affect) the conceptualization.

Here, we also discuss here the two most recent tools related smell detection and software visualization. In (Murphy-Hill and Black 2010), Murphy-Hill and Black presented a visualization implemented as an Eclipse plug-in. The tool is composed of sectors in a semicircle on the right-hand side of the editor pane, called petals: each petal corresponds to a smell. They performed a controlled experiment with 12 participants (6 programmers and 6 students) to evaluate the tool. Their main findings were: i) programmers identify more smells using the tool than not using the tool ii) smells are subjective and iii) the tool helps in deciding. Carneiro et al. (2010) presented the SourceMiner tool; a multi-perspective environment Eclipse based plug-in. SourceMiner has visualizations that address inheritance and coupling characteristics of a program. The visualizations also portray the previously mapped concerns of the analyzed software. The authors performed an exploratory study with five developers, using a concern mapping multi-perspective approach to identify code smells. Two main findings were presented. First, the concern visualizations provided useful support to identify God Class and Divergent Change smells. Second, strategies for smell detection supported by the multiple concern views were revealed.

Smells and human aspects

Mäntylä (2005) presents results of two experiments addressing agreement in smell detection and factors to explain it. A small application in Java with nine classes and 1000 LOC was created and used in both experiments. In the first experiment, there were three questions about "Long Method", "Long parameter List" and "Feature Envy" smells, and one question asked if the method should be refactored to remove the detected smells. In the second experiment, participants were only asked if some specific methods should be refactored. He found high rates of agreement for simple smells: long method and long parameter list. He found weaker agreement levels, however, concerning the feature envy smell and the decisions regarding refactoring. Mäntylä then tried to identify factors that influenced agreement in smell detection. He investigated both the influence of software metrics and demographic data as factors for smell detection agreement. His findings point to the influence of metrics.

Mäntylä and Lassenius (2006a) investigated why and when people think a code needs refactoring. They analyzed one of the experiments presented in (Mäntylä 2005) to investigate what drivers define the refactoring decisions. They applied a questionnaire to

understand refactoring decisions. A taxonomy was defined after a qualitative analysis (coding process) of the textual answers. The authors also compared the results with an automatic detection tool. The most important driver was the size of a method. One of their important findings was that there was a conflict of opinions between the participants. The conflict was related to the assessed internal quality of the methods and the need to refactor them. Regarding the automatic detection, they found that some drivers are difficult or impossible to detect automatically, and some smells are better detected by experienced participants than by automatic means.

Schumacher et al. (2010) build on and extend Mäntylä and Lassenius's (2006a) work. They investigated the way professional software developers detect god class smells, and then compared these results to automatic classification. The study was done in a professional environment, with two real projects and two participants in each project. The research questions focused on "Evaluation of Human Performance" and "Evaluation of Automatic Classifiers". Participants were introduced to the god class smell in a short presentation and were asked to detect them in specific code pieces. During this task, they received a list of questions to help with the identification of god classes (the support questions adopted by us in this work), questions such as: "Does the class have more than one responsibility?". A process was designed to ensure that all participants performed the inspection of classes in a similar fashion. To evaluate the participant performance in the task, they used a "think-aloud" protocol (recorded as audio) and data collection forms. Coding was carried out to identify drivers and answers from the data collection form were used to evaluate time and agreement. Their main findings were: (1) there was low agreement among participants and (2) "misplaced method" was the strongest driver for god class detection. Related to the evaluation of automatic detection, their main findings were: (1) an automated metric-based pre-selection decreases the effort needed for manual code inspections and (2) automatic detection followed by manual review increases the overall confidence.

A study with aims similar to Schumacher's was presented by (Mäntylä et al. in 2004 and Mäntylä and Lassenius 2006b). Through a survey, they asked participants about 23 smells and used a scale from 1 (lack) to 7 (large presence) to evaluate the presence of smells in a piece of code. They received 12 completed questionnaires from 18 sent, all being sent to developers in a small software company. In one of the findings the authors declare: "*the use of smells for code evaluation purposes is hard due to conflicting perceptions of different evaluators*".

It is important to note that these studies focused on specific human aspects. Basically, they investigated agreement and decision drivers adopted by participants. We built on the discussions considering two important aspects: the strategies adopted and the impact of visualization on each investigated aspect. Moreover, we proposed a discussion about how each of these aspects reflects the problem of conceptualization. We consider this our main contribution.

Conclusions

The purpose of this work was to find empirical evidence to evaluate the impact of personal conceptualization in god class detection. We were interested in understanding how differences in the perception of the concept affect identification. To do this, we performed a controlled experiment that extended another study focusing on investigating

how developers detect god classes. Our experiment deepened and detailed some research questions previously presented and added new research questions. We addressed agreement among participants, and also among participants and an oracle, decision drivers, the impact of using a visualization tool, and strategies adopted by participants in god class detection. Our analysis considered how these elements are more related to personal choice than to the conceptual aspects in god class detection.

Our main finding is that the problem of god class detection is mainly related to conceptualization, i.e., agreeing on what a god class is, and which thresholds should be adopted. We believe that this type of problem is not solved by using other observation approaches, such as proposing a new metric or a new visual resource. We also believe this issue is transversal to other code smells. The smell detection problem would be better addressed if the community identifies concrete "good" examples of smells, providing standard definitions of them, teaching people about their conceptualization, and only then providing tools and methods (using metrics, thresholds or visualizations) to aid in their identification. Another important finding was that our work produced low agreement rates in code smell detection among the experiment participants, which is in accordance with other works.

To address the limitations of this study and to further develop it in this area, we are planning to replicate the experiment with more experienced participants to evaluate the impact of experience on the process. Other aspects that we may replicate as well is the evaluation of other software and other smells. To support replication we provide the experimental package[d]. The package contains forms, data and software.

Endnotes

[a]Eclipse IDE - http://www.eclipse.org/downloads/; Usage Data Collector (UDC) plug-in - http://www.eclipse.org/epp/usagedata/; Task Register - private; SourceMiner - visual support http://www.sourceminer.org/

[b]Web address of ProM tool: www.processmining.org

[c]Fleiss Kappa is a Cohen's Kappa variation that permits test with more than two raters

[d]Experimental package: http://wiki.dcc.ufba.br/LES/FindingGdoClassExperiment2012

Competing interests
The authors declare that they have no competing interests.

Authors' contributions
JAMS and MGM planned, performed and analyzed the experiment and drafted the manuscript. CPS supported some specific analysis. RLN also supported some analysis and drafted the manuscript. All authors read and approved the final manuscript.

Acknowledgements
We would like to thanks: Claudio Sant'Anna for allowing execution of the experiment in the Software Quality course; Bruno Carneiro for valuable participation in definition of oracle; and to participants for their availability and effort.

Author details
[1]Department of Technology, State University of Feira de Santana, Transnordestina avenue S/N - Feira de Santana - Bahia, Feira de Santana, Brazil. [2]Mathematics Institute, Federal University of Bahia, Ademar de Barros Avenue, S/N, Salvador - Bahia, Salvador, Brazil. [3]Fraunhofer Project Center for Software & Systems Eng., Ademar de Barros Avenue, S/N, Salvador - Bahia, Salvador, Brazil. [4]Information Technology Department, Federal Institute of Bahia, Araujo Pinho Avenue, 39, Salvador - Bahia, Salvador, Brazil.

Received: 19 December 2013 Accepted: 31 August 2014
Published online: 18 September 2014

References

Abbes M, Khomh F, Guéhéneuc Y-G, Antoniol G (2011) An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In: Proc. of 15th European Conference on Software Maintenance and Reengineering (CSMR). IEEE, Oldenburg, Germany, pp 181–190

Carneiro GF, Mendonça MG (2013) Sourceminer: A multi-perspective software visualization environment. In: Proceedings of 15th International Conference on Interprise Information Systems. ICEIS. SciTePress, Angers, France

Carneiro G, Silva M, Maia L, Figueiredo E, Sant'Anna C, Garcia A, Mendonça M (2010) Identifying code smells with multiple concern views. In: Proc. of 1th Brazilian Conference on Software: Theory and Practice, CBSOFT. IEEE, Salvador, Bahia, Brazil

Dybå T, Sjøberg DIK, Cruzes DS (2012) What works for whom, where, when, and why?: On the role of context in empirical software engineering. In: Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. ESEM '12. ACM, New York, NY, USA, pp 19–28

Feinstein AR, Cicchetti DV (1990) High agreement but low kappa: I. the problems of two paradoxes. J Clin Epidemiol 43(6):543–549

Finn RH (1970) A note on estimating the reliability of categorical data. Educ Psychol Meas 30:71–76

Fleiss JL (1971) Measuring nominal scale agreement among many raters. Psychol Bull 76(5):378–382

Fowler M (1999) Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA

Fontana FA, Mariani E, Morniroli A, Sormani R, Tonello A (2011) An experience report on using code smells detection tools. In: Proc. of 4th Software Testing, Verification and Validation Workshops, ICSTW. IEEE, Berlin, Germany

Gwet K (2002) Kappa statistic is not satisfactory for assessing the extent of agreement between raters. Stat Methods Inter-rater Reliability Assess 1:1–5

Höst M, Regnell B, Wohlin C (2000) Using students as subjects— a comparative study of students and professionals in lead-time impact assessment. Empirical Softw Eng 5(3):201–214

Höst M, Wohlin C, Thelin T (2005) Experimental context classification: incentives and experience of subjects. In: Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference On. IEEE, St Louis, Missouri, USA, pp 470–478

Johnson B, Shneiderman B (1991) Tree-maps: a space-filling approach to the visualization of hierarchical information structures. In: Visualization, 1991. Visualization '91, Proceedings., IEEE Conference On. IEEE, San Diego, CA, USA, pp 284–291

Landis JR, Koch GG (1977) The measurement of observer agreement for categorical data. Biometrics 33(1):159–174

Lanza M, Ducasse S (2003) Polymetric views - a lightweight visual approach to reverse engineering. Softw Eng IEEE Trans 29(9):782–795

Lanza M, Marinescu R, Ducasse S (2005) Object-Oriented Metrics in Practice. Springer, Secaucus, NJ, USA

Li W, Shatnawi R (2007) An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. J Syst Softw 80(7):1120–1128

Meyer B (1988) Object-Oriented Software Construction, 1st edn. Prentice-Hall, Inc., Upper Saddle River, NJ, USA

Moha N, Gueheneuc Y-G, Duchien L, Le Meur A-F (2010) Decor: A method for the specification and detection of code and design smells. IEEE Trans Softw Eng 36(1):20–36

Murphy-Hill E, Black AP (2010) An interactive ambient visualization for code smells. In: Proc. of the 5th ACM Symposium on Software Visualization, SOFTVIS. ACM, Salt Lake City, Utah, USA

Mäntylä M (2005) An experiment on subjective evolvability evaluation of object-oriented software: explaining factors and interrater agreement. In: Proc. of the 4th International Syimposium on Empirical Software Engineering, ISESE. IEEE, Noosa Heads, Australia

Mäntylä MV, Lassenius C (2006a) Drivers for software refactoring decisions. In: Proceedings of the International Symposium on Empirical Software Engineering, ISESE. ACM, Rio de Janeiro, Brazil

Mäntylä M, Lassenius C (2006b) Subjective evaluation of software evolvability using code smells: An empirical study. Empirical Softw Eng 11(3):395–431

Mäntylä M, Vanhanen J, Lassenius C (2004) Bad smells - humans as code critics. In: 20th IEEE International Conference on Software MaintenanceICSM 2004, ICSM. IEEE, Chicago Illinois, USA

Olbrich S, Cruzes DS, Basili V, Zazworka N (2009) The evolution and impact of code smells: a case study of two open source systems. In: Proc. of the 3rd International Symposium on Empirical Software Engineering and Measurement, ESEM. IEEE, Lake Buena Vista, Florida, USA

Olbrich SM, Cruzes DS, Sjoberg DIK (2010) Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems. In: Proc. of the IEEE International Conference on Software Maintenance, ICSM. IEEE, Timisoara, Romania

Padilha J, Figueiredo E, Sant'Anna C, Garcia A (2013) Detecting god methods with concern metrics: An exploratory study. In: Proceedings of the 7th Latin-American Workshop on Aspect-Oriented Software Development(LA-WASP), Co-allocated with CBSoft. IEEE, Brasília, Brazil

Parnin C, Görg C, Nnadi O (2008) A catalogue of lightweight visualizations to support code smell inspection. In: Proc. of the 4th Software Visualization, SOFTVIS. ACM, Herrsching am Ammersee, Germany

Powers DMW (2012) The Problem with Kappa. In: Proceedings of the 13th Conference of the European Chapter of the Association for Computational Linguistics, EACL '12, Stroudsburg, PA, USA, pp 345–355

Rapu D, Ducasse S, Girba T, Marinescu R (2004) Using history information to improve design flaws detection. In: Proc. of 8th European Conference on Software Maintenance and Reengineering, CSMR. IEEE, Tampere, Finland

Riel AJ (1996) Object-Oriented Design Heuristics, 1st edn. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA

Santos JA, Mendonça M, Silva C (2013) An exploratory study to investigate the impact of conceptualization in god class detection. In: Proc of 17th International Conference on Evaluation and Assessment in Software Engineering, EASE. ACM, Porto de Galinhas, Brazil

Schumacher J, Zazworka N, Shull F, Seaman C, Shaw M (2010) Building empirical support for automated code smell detection. In: Proc. of the International Symposium on Empirical Software Engineering and Measurement, ESEM. ACM, Bolzano-Bozen, Italy

Simon F, Steinbruckner F, Lewerentz C (2001) Metrics based refactoring. In: Proc. of 5th European Conference on Software Maintenance and Reengineering, CSMR. IEEE, Lisbon, Portugal

Sjøberg DIK, Yamashita A, Anda BCD, Mockus A, Dyba T (2013) Quantifying the effect of code smells on maintenance effort. IEEE Trans Softw Eng 39(8):1144–1156

van der Aalst WMP (2011) Process mining: discovery, conformance and enhancement of business processes. 1st edn., p. 352. Springer, Berlin

Van Emden E, Moonen L (2002) Java quality assurance by detecting code smells. In: Proc. of the 9th Working Conference on Reverse Engineering, WCRE. IEEE, Washington, DC, USA

Vinson NG, Singer Ja (2008) A practical guide to ethical research involving humans. In: Shull F, Singer J, Søberg DIK (eds) Guide to Advanced Empirical Software Engineering. Springer, London, pp 229–256

Whitehurst GJ (1984) Interrater agreement for journal manuscript review. Am Psychol 39(1):22–28

Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A (2012) Experimentation in Software Engineering. p 250

Zhang M, Hall T, Baddoo N (2011) Code bad smells: A review of current knowledge. J Softw Maint Evol 23(3):179–202

doi:10.1186/s40411-014-0011-9
Cite this article as: Santos *et al.*: The problem of conceptualization in god class detection: agreement, strategies and decision drivers. *Journal of Software Engineering Research and Development* 2014 **2**:11.